Tutorial 3

Realizzazione di un generatore di segnali sinusoidali su DE1

Descrizione: La DE1 monta un DECODER a frequenze audio, programmabile tramite il protocollo I2C. Si deve inizialmente realizzare un circuito atto alla configurazione del decoder e successivamente, utilizzando funzioni precostituite si realizzi un generatore di segnali sinusoidali e lo si interfacci al precedente DECODER.

Scopo: Approfondimento del linguaggio Verilog, gestione e controllo di protocolli di comunicazione, utilizzo di blocchi funzionali costruiti da terze parti, utilizzo di strumenti per il debugging fisico di un sistema operante su FPGA.

Apprendimenti previsto:

- Sviluppo di un sistema di comunicazione I2C
- Testing "on board" tramite "Signal Tap II _ Logic Analizer"
- Utilizzo delle "mega core functions"
- Conversione Parallelo-seriale

Procedimento:

Si inizi un nuovo progetto per `Ciclone II - EP2C20F484C7N`

Realizzazione di un generatore di segnali I2C

Si realizzi un blocco atto a generare i segnali I2C. In ingresso al blocco oltre al clock ed al reset (sul fronte negativo) vi sia il dato da trasmettere (a 16 bit) un linea "go" atta a confermare la congruità del segnale in ingresso ed una linea "ack" che viene interrogata ogni nove bit di trasmissione per verificare se il dispositivo ricevente abbia o meno ricevuto il messaggio di "*ricevuto*" ovvero "*acknoledge*" . In uscita siano presenti oltre i segnali *SDA* ed *SCL* propri del protocollo I2C un segnale "*data_rdy*" che segnala che il dispositivo è pronto a ricevere nuovi valori in ingresso ed una linea di "*error*" che viene attivata se il ricevitore non accusi "*ricevuto*" alla fine della trasmissione del messaggio.

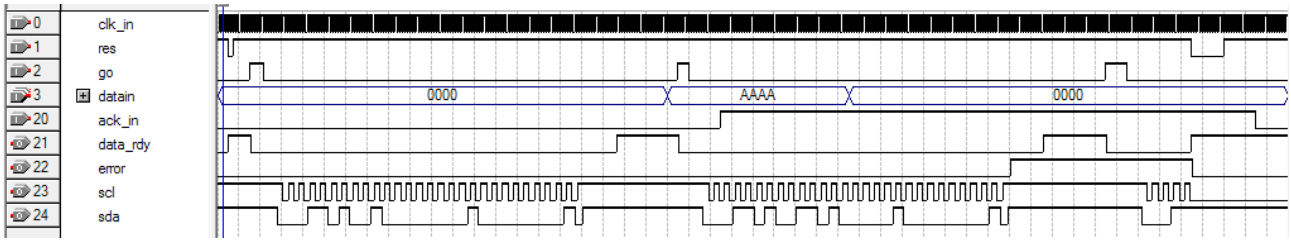
Una realizzazione del modulo potrebbe essere la seguente:

```
module gen_i2c (clk_in,res,datain,ack_in,go,sda,scl,data_rdy,error);

input   clk_in;
input    res;
input    [15:0] datain;
input    go;
input    ack_in;
output  reg sda=1;
output  reg scl=1;
output   reg data_rdy;
output   reg error;

reg [8:0] counter;
reg [3:0] cmd_sda;
reg [3:0] cmd_scl;
reg ACK1,ACK2,ACK3;

wire [1:0] counter_4=counter[1:0];
wire [6:0] counter_bit=counter[8:2];

reg Trans_ON;
wire [23:0] bits={8'h34,datain};
```

```
// Trans_ON segnale per indicare il periodo di trasmissione dati attiva
always @(posedge clk_in or negedge res) begin
if (!res)
    begin; Trans_ON=0; data_rdy=1; end
else if (go)
    begin; Trans_ON=1; data_rdy=0; end
else if (counter_bit ==7'd33)
    begin; Trans_ON=0; data_rdy=1; end
end


// generazione dei segnali
always @(posedge clk_in or negedge res or posedge go) begin
if (!res|go) counter=0;
else if (Trans_ON)
begin
  counter=counter+1;
  case (counter_bit)
        6'd0  : begin cmd_sda=4'b1111; cmd_scl=4'b1111; end
        // start
        6'd1  : begin cmd_sda=4'b0001; cmd_scl=4'b0111; end
        // device
        6'd2  : begin cmd_sda={bits[23],bits[23],bits[23],bits[23]}; cmd_scl=4'b0110; end
        6'd3  : begin cmd_sda={bits[22],bits[22],bits[22],bits[22]}; cmd_scl=4'b0110; end
        6'd4  : begin cmd_sda={bits[21],bits[21],bits[21],bits[21]}; cmd_scl=4'b0110; end
        6'd5  : begin cmd_sda={bits[20],bits[20],bits[20],bits[20]}; cmd_scl=4'b0110; end
        6'd6  : begin cmd_sda={bits[19],bits[19],bits[19],bits[19]}; cmd_scl=4'b0110; end
        6'd7  : begin cmd_sda={bits[18],bits[18],bits[18],bits[18]}; cmd_scl=4'b0110; end
        6'd8  : begin cmd_sda={bits[17],bits[17],bits[17],bits[17]}; cmd_scl=4'b0110; end
        6'd9  : begin cmd_sda={bits[16],bits[16],bits[16],bits[16]}; cmd_scl=4'b0110; end
        // ACK1
        6'd10  : begin cmd_sda=4'b1111; cmd_scl=4'b0110; end
        // address
        6'd11  : begin cmd_sda={bits[15],bits[15],bits[15],bits[15]}; cmd_scl=4'b0110; end
        6'd12  : begin cmd_sda={bits[14],bits[14],bits[14],bits[14]}; cmd_scl=4'b0110; end
        6'd13  : begin cmd_sda={bits[13],bits[13],bits[13],bits[13]}; cmd_scl=4'b0110; end
        6'd14  : begin cmd_sda={bits[12],bits[12],bits[12],bits[12]}; cmd_scl=4'b0110; end
        6'd15  : begin cmd_sda={bits[11],bits[11],bits[11],bits[11]}; cmd_scl=4'b0110; end
        6'd16  : begin cmd_sda={bits[10],bits[10],bits[10],bits[10]}; cmd_scl=4'b0110; end
        6'd17  : begin cmd_sda={bits[9],bits[9],bits[9],bits[9]}; cmd_scl=4'b0110; end
        6'd18  : begin cmd_sda={bits[8],bits[8],bits[8],bits[8]}; cmd_scl=4'b0110; end
        // ACK2
        6'd19  : begin cmd_sda=4'b1111; cmd_scl=4'b0110; end
        // data
        6'd20  : begin cmd_sda={bits[7],bits[7],bits[7],bits[7]}; cmd_scl=4'b0110; end
        6'd21  : begin cmd_sda={bits[6],bits[6],bits[6],bits[6]}; cmd_scl=4'b0110; end
        6'd22  : begin cmd_sda={bits[5],bits[5],bits[5],bits[5]}; cmd_scl=4'b0110; end
        6'd23  : begin cmd_sda={bits[4],bits[4],bits[4],bits[4]}; cmd_scl=4'b0110; end
        6'd24  : begin cmd_sda={bits[3],bits[3],bits[3],bits[3]}; cmd_scl=4'b0110; end
        6'd25  : begin cmd_sda={bits[2],bits[2],bits[2],bits[2]}; cmd_scl=4'b0110; end
        6'd26  : begin cmd_sda={bits[1],bits[1],bits[1],bits[1]}; cmd_scl=4'b0110; end
        6'd27  : begin cmd_sda={bits[0],bits[0],bits[0],bits[0]}; cmd_scl=4'b0110; end
        // ACK3
        6'd28  : begin cmd_sda=4'b1111; cmd_scl=4'b0110; end
        // stop
        6'd29  : begin cmd_sda=4'b1000; cmd_scl=4'b1110; end
        // end
        6'd30  : begin cmd_sda=4'b1111; cmd_scl=4'b1111; end
  endcase
  sda=cmd_sda[counter_4];
  scl=cmd_scl[counter_4];
end
end

// verifica ACK
always @(posedge clk_in)
if (!res) begin ACK1=1;ACK2=1;ACK3=1;error=0;end
else
begin
if (counter_bit==6'd10 & counter_4==3) ACK1=ack_in;
if (counter_bit==6'd19 & counter_4==3) ACK2=ack_in;
if (counter_bit==6'd28 & counter_4==3) ACK3=ack_in;
if (counter_bit > 6'd29) error=ACK1&ACK2&ACK3;
end
endmodule
```

Forse il file proposto non risulta propriamente “elegante” da un punto di vista strettamente estetico e potrebbe essere riadattato in modo da essere più compatto e fors’anche più leggibile. Va peraltro sottolineato che non sempre (anzi, quasi mai) una stesura elegante del codice dal punto di vista della “compattezza” viene poi sintetizzata in modo efficace a livello circuitale.

- Si setti temporaneamente il file come “top Level Entity”
- Si generi un opportuno file di stimoli
- Si simuli funzionalmente il precedente modulo ne si verifichi il corretto funzionamento.



Si noti ad esempio:

- che il dato in ingresso non deve essere modificato se prima la linea *data_rdy* non ritorna allo stato alto (altrimenti il messaggio trasmesso perde di congruità)
- che i primi 8 bit competono all’indirizzo del dispositivo (h34) mentre gli altri 16 sono inerenti al dato da trasmettere
- la presenza di errore se la linea *ack_in* non è bassa in presenza del nono bit trasmesso.
- Che il simulatore, NON disponendo di un dispositivo reale capace di interagire con la linea SDA in tri-state, debba richiedere la presenza di un ulteriore segnale *ack_in* (al momento pilotato dall’utente) per rilevare ipotetici errori di trasmissione.

Realizzazione di un blocco che fornisca la sequenza I2C

Si realizzi un blocco che interagendo coi segnali del blocco sovraesposto fornisca con la corretta sequenza i segnali da far giungere al DECODER. Il sistema riceve in ingresso oltre il `clock` ed il `reset` il segnale `data_rdy` col quale rileva che il dispositivo ricevente è pronto a ricevere i dati. In uscita fornisce il dato da trasmettere ed il segnale di attivazione “`go`”. Una soluzione potrebbe essere la seguente:

```
`define rom_size 6'd10

module gen_code (clk,res,rdy_in,data,go );
        input  clk,res,rdy_in;
        output reg [15:0] data;
        output reg go;

        reg [3:0] addr;
        reg [15:0]ROM[`rom_size:0];
        reg go1,go_pulse;

// generazione segnale "go"
always @(posedge clk or negedge res) begin
    if (!res)  go=0;
        else if ((rdy_in) & (addr < `rom_size)) go=1;
        else go=0;
end

// generazione di un impulso unitario in corrispondenza al fronte di salita
always @(posedge clk)  go1=go;
always @(posedge clk) begin if (!go1 & go) go_pulse=1; else go_pulse=0; end
```

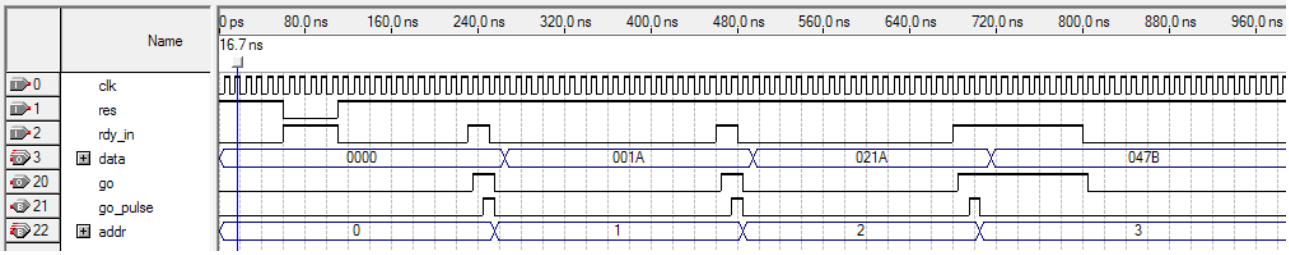
```
// incremento dell'indirizzo
always @(posedge clk or negedge res) begin
    if (!res) addr = 0;
    else if (addr < `rom_size & go_pulse) addr=addr+1; end

//dato in uscita
always @(posedge clk)
       begin
       ROM[0]= 16'h0000;            // dummy
       ROM[1]= 16'h001A;            // linvol
       ROM[2]= 16'h021A;            // rinvol
       ROM[3]= 16'h047B;            // lhpvol
       ROM[4]= 16'h067B;            // rhpvol
       ROM[5]= 16'h0810;            // audiopath
       ROM[6]= 16'h0a06;            // digitalpath - deenfasi
       ROM[7]= 16'h0c00;            // power down disable
       ROM[8]= 16'h0e02;            // format and master
       ROM[9]= 16'h1000;            // control
       ROM[10]= 16'h1201;           // active

       data=ROM[addr];
       end
endmodule
```

Da notare in particolare che tutti i segnali sono sincronizzati sul medesimo clock onde prevenire problemi di disallineamento del clock. In particolare, poiché il segnale "`go`" può perdurare per più cicli di clock, un suo impiego diretto per controllare l'indirizzo della ROM è sconsigliato infatti:

- Se si sincronizza il sistema sul FRONTE del segnale "go" si rischia un disallineamento dei clock
- Se si sincronizza il sistema sullo STATO del segnale "go" si rischia che il conteggio non si limiti ad incrementare di una sola unità l'indirizzo, ma che questo si estenda per più cicli di clock

Il problema è stato ovviato creando un segnale opportuno (`go_pulse`) che dura un solo ciclo di clock e che può essere impiegato per sincronizzare il conteggio dell'indirizzo della rom.

Si verifichi tramite opportuna simulazione funzionale il corretto funzionamento del modulo.



Nota: per accedere a segnali che non siano solamente quelli relativi alle porte di I/O del modulo, all'interno della finestra "node Finder" si faccia riferimento ai segnali disponibili quando si predispone il filtro a: *Design Entry (all names)*

Si noti come in pratica il dato all'indirizzo "0" non possa essere mai letto, infatti quando viene generato il segnale "go" il dato cambia subito dopo. Pertanto all'indirizzo 0 viene messo un dato "dummy".

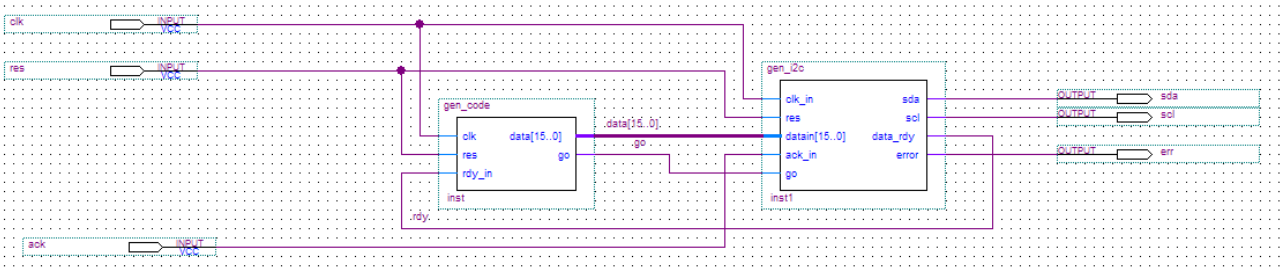
Realizzazione del sistema completo di configurazione I2C

Poiché il protocollo I2C prevede una frequenza di trasmissione di qualche decina di KHz bisogna predisporre ancora un blocco atto a ridurre opportunamente la frequenza di clock. Una possibile realizzazione è la seguente:

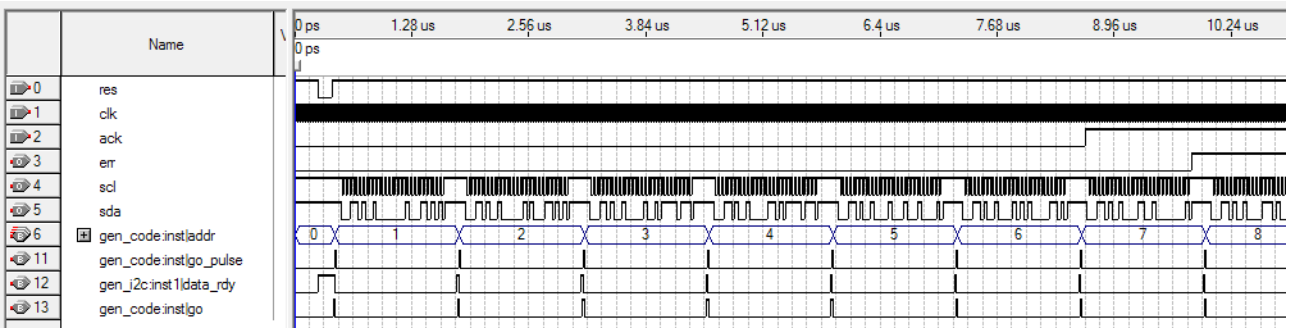
```
module clk_div(clk_in,res,clk_i2c);

input clk_in,res;
output reg clk_i2c;

reg [12:0] div_i2c;

//      Clock Setting

parameter       CLK_Freq        =       50000000;       //      50      MHz
parameter       I2C_Freq        =       20000;          //      20      KHz

////////////// I2c clk /////////////////
always@(posedge clk_in or negedge res)
begin
        if(!res)
        begin
                div_i2c <=      0;
                clk_i2c <=      0;
        end
        else
        begin
                if( div_i2c     < ((CLK_Freq/(I2C_Freq*2))-1) )
                div_i2c <=      div_i2c+1;
                else
                begin
                        div_i2c <=      0;
                        clk_i2c <=      ~clk_i2c;
                end
        end
end

endmodule
```

Per i tre moduli così realizzati si costruisca il simbolo corrispondente.

Si noti che ove si desideri simulare lo schematico completo è meglio soprassedere all'impiego del riduttore di frequenza, ed altresì la linea `ack_in` deve essere pilotata manualmente dall'utente onde emulare la presunta risposta del decoder. Notare inoltre l'impiego del segnale `data_rdy` in loop

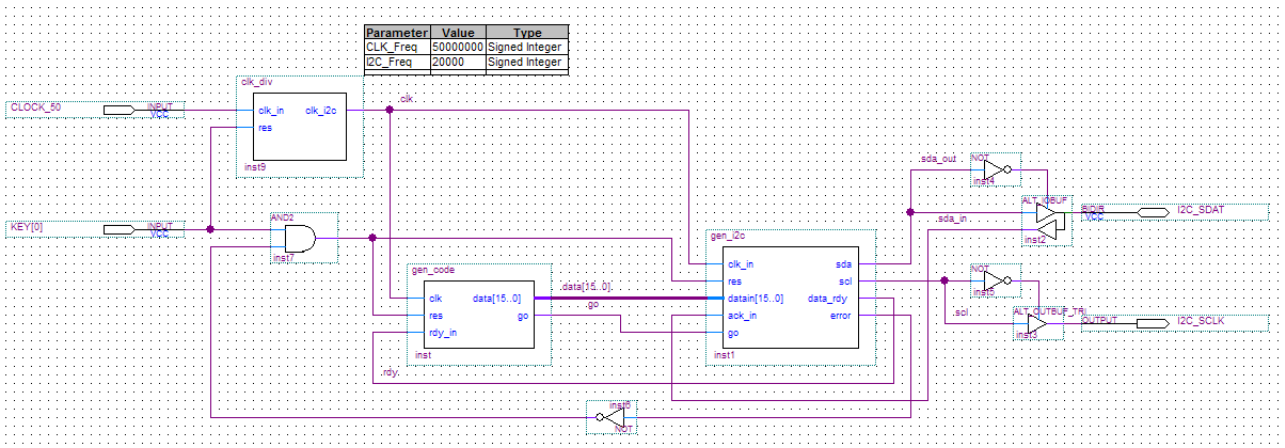


Generando un opportuno file di stimoli si può verificare tramite simulazione funzionale il funzionamento:



Lo schema completo deve peraltro evidenziare oltre al collegamenti tra i vari blocchi, le porte di controllo e le corrispondenti uscite ed i nomi di questi devono rispecchiare i nomi del file di vincoli. Si tenga inoltre in particolar conto il fatto che i segnali d'uscita SDA ed SCL possono assumere o lo stato basso (0) oppure lo stato di alta impedenza (Z), che poi assume il valore logico alto grazie alla resistenza di "pullup" integrata sulla scheda DE1 e che la porta per SDA deve essere di tipo bidirezionale. Inoltre il segnale di errore può essere utilizzato, dopo essere stato invertito per "resettare" il sistema.

Lo schema completo potrebbe essere il seguente:

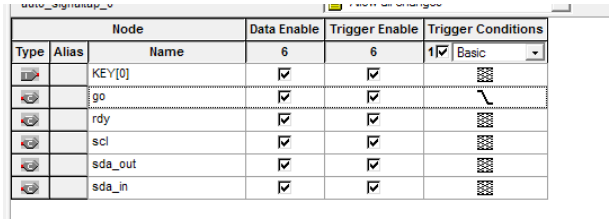


Questo sistema non può essere simulato poiché mancante del segnale di `ACK` che dovrebbe pervenire dal decoder. Si può però verificare il corretto funzionamento direttamente sulla scheda.

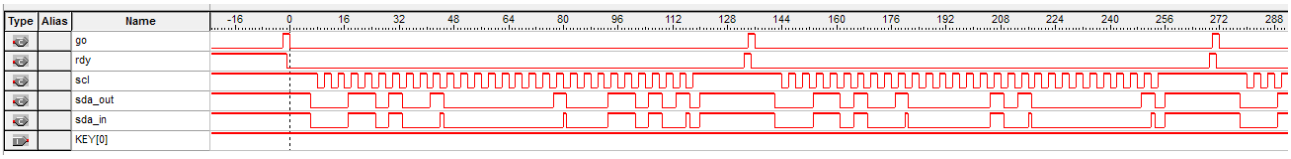
Testing su Scheda

- Si effettui una compilazione parziale del progetto
  - `Start Analysis and Synthesis`
- Si aggiunga al progetto un nuovo file di tipo Signal Tap II – Logic Analizer
  - `File > New - Signal Tap II - Logic Analizer`
- Si definisca un clock sul quale sincronizzare l'acquisizione dei dati (si consiglia di usare il clock a bassa frequenza piuttosto che il clock a 50 MHz). Per fare ciò è consigliabile nominare opportunamente la corrispondente "net" sullo schematico.
  - Cliccare su "…" nella scheda Signal Configuration – clock
  - Filtrare tramite "Signal Tap II – Pre Sinthesis"
  - Scegliere il clock
- Si definisca una profondità di memoria di 2K
  - Sample depth = 2K

- Si definisca quali segnali monitorare (per identificarli comodamente risulta comodo renderli univoci dando essi un nome all'interno dello schematico)
    - Edit > Add Node
    - Filtrare tramite "Signal Tap II – Pre Sinthesis"
    - Scegliere i segnali da monitorare
- Si definisca su quale trigger sincronizzare l'acquisizione
  (ad esempio fronte negativo del segnale `go`)
    - Right click on Trigger condition
    - Scegliere l'opzione desiderata



- Si salvi il file così realizzato.
- Si compili il progetto completo – ricordandosi di imporre i vincoli corretti.
- Si effettui il download su scheda
- Nella Finestra JTAG Chain Configuration verificare che l'Hardware coincida con l'USB Blaster e che il dispositivo rilevato coincida con l'FPGA montata sulla DE1.
- Si inizi l'analisi dei segnali
    - Processing > Run Analysis
- Si pigi il tasto preposto al reset sulla scheda (Key[0])
- Si verifichino le forme d'onda in uscita



Si noti in particolare come la forma d'onda rilevata in ingresso su SDA coincida con quella in uscita, fatta eccezione per ogni nono bit, momento nel quale il decoder segnala l'ACK abbassando detta linea durante tutto il periodo per il quale SCL rimane alto.

Si provi ora a modificare l'indirizzo del dispositivo e si verifichi che quest'ultimo, non riconoscendo la trasmissione come a lui diretta, non segnali alcun ACK! Questo crea un segnale d'errore che resetta continuamente il sistema ritentando la trasmissione all'infinito.

Configurazione del Decoder

Facendo riferimento agli 11 registri del decoder, provare a configurarlo secondo varie metodologie.

REGISTER	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B8	B7	B6	B5	B4	B3	B2	B1	B0
R0 (00h)	0	0	0	0	0	0	0	LRIN BOTH	LIN MUTE	0	0	LINVOL				
R1 (02h)	0	0	0	0	0	0	1	RLIN BOTH	RIN MUTE	0	0	RINVOL				
R2 (04h)	0	0	0	0	0	1	0	LRHP BOTH	LZCEN	LHPVOL						
R3 (06h)	0	0	0	0	0	1	1	RLHP BOTH	RZCEN	RHPVOL						
R4 (08h)	0	0	0	0	1	0	0	0	SIDEATT		SIDETONE	DAC SEL	BY PASS	INSEL	MUTE MIC	MIC BOOST
R5 (0Ah)	0	0	0	0	1	0	1	0	0	0	0	HPOR	DAC MU	DEEMPH		ADC HPD
R6 (0Ch)	0	0	0	0	1	1	0	0	PWR OFF	CLK OUTPD	OSCPD	OUTPD	DACPD	ADCPD	MICPD	LINEINPD
R7 (0Eh)	0	0	0	0	1	1	1	0	BCLK INV	MS	LR SWAP	LRP	IWL		FORMAT	
R8 (10h)	0	0	0	1	0	0	0	0	CLKO DIV2	CLKI DIV2	SR				BOSR	USB/ NORM
R9 (12h)	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	ACTIVE
R15(1Eh)	0	0	0	1	1	1	1	RESET								
	ADDRESS							DATA								

- Collegare una sorgente sonora in ingresso ed un sistema per l'ascolto (casse-cuffia) in uscita
- Provare a modificare il volume in ingresso e/ o in uscita
- Provare a modificare la sorgente attivando solo il convertitore DAC– escludere sidetone, bypass e configurare linein per la linea in ingresso (h0810)
- Introdurre un blocco che ricevendo in ingresso un clock a 50 MHz ne generi uno a 12,5 MHz e si usi quest'ultimo per pilotare il master clock del decoder. Si può impiegare il medesimo blocco realizzato prima, pur di modificarne i parametri (basta agire esclusivamente a livello di schematico) senza modificare il sorgente verilog.
- Successivamente configurare il decoder come Master (esso genera tutti i segnali) (h0E41)

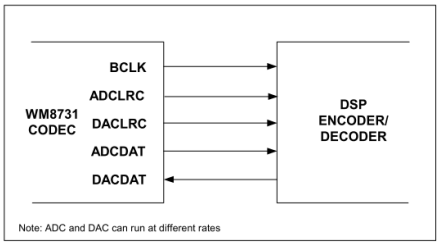
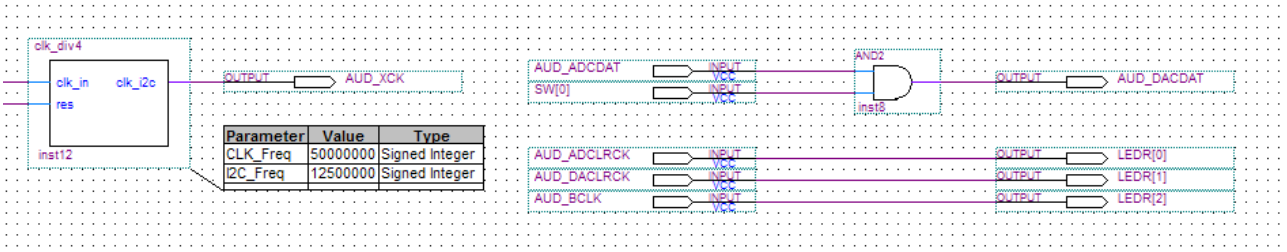
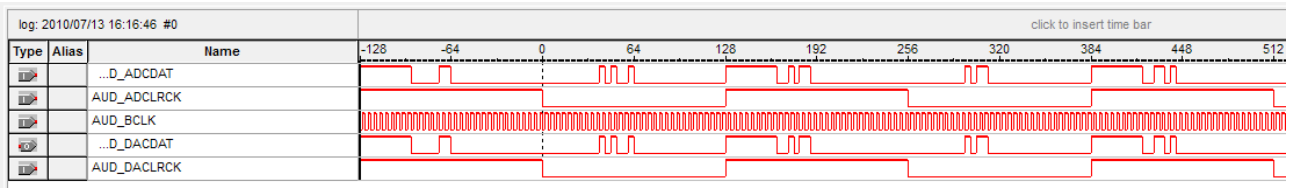


Figure 30 Master Mode

- Si aggiunga allo schematico la seguente configurazione



- Si provi ad utilizzare il "Signal Tap II – Logic Analizer" per monitorare i segnali audio generati dal decoder, configurando il clock di acquisizione su AUD_XCK



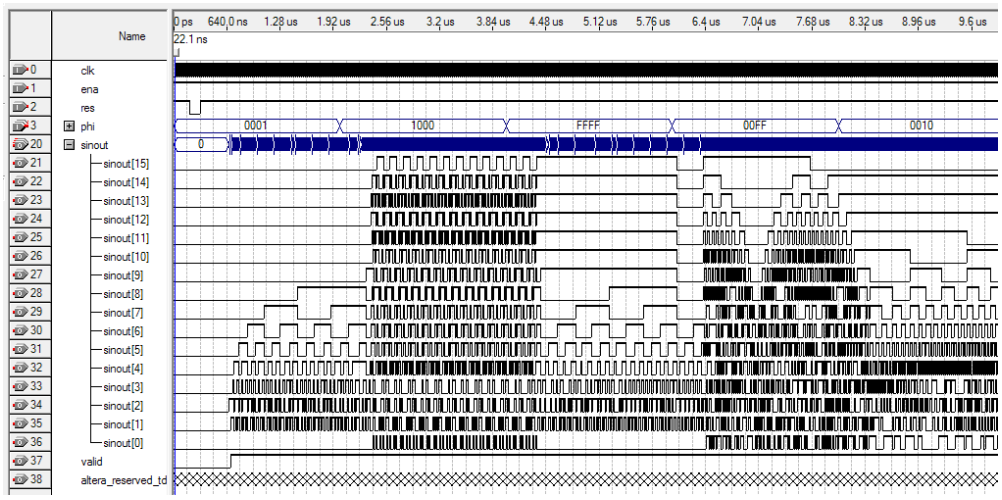
Generazione di un segnale sinusoidale

Si utilizzino le "megafunction" una serie di blocchi sviluppati da terze parti:

- `Tool > Mega Wizard Pulgin Manager`
- `Create a New custom Megafunction Variation`
- Si scelga `DSP – Signal Generator – NCO v9.1`
- Si assegni un nome opportuno, il linguaggio di implementazione (Verilog) e la famiglia di FPGA (Ciclone II)
- Nella parametrizzazione del modulo (Step 1) **:**
  - Nella Cartella Parameters
    - Si fissi a 16 bit: Phase accumulator precision, Angular resolution, Magnitude Precision.
    - NO implement phase dithering
    - Si scelga la tipologia di algoritmo (Cordic) da implementare e si visualizzino le risorse impiegate (nell'opportuna sotto-cartella)

    NOTA: la frequenza di clock in ingresso e la frequenza desiderata in uscita non modificano l'implementazione del blocco ma aiutano a calcolare il valore da fornire in ingresso al blocco stesso per realizzare la frequenza desiderata.
  - Nella cartella implementation si predisponga
    - Un solo canale (single Output)
    - NO frequency modulation Input
    - NO Phase Modulation Input
    - Cordic Implementation Parallel
- Nella setup Simulation (Step 2)
  - Generate Simulation Model
- Generate
  - A Processo ultimato : Exit
  - alla domanda: Do you want to add IP to the project: rispondere YES

Ora all'interno del progetto vi è il blocco generato, che potrebbe, dopo essere stato importato in un opportuno schematico e generando un opportuno file di stimoli, essere simulato.



Realizzazione di un blocco per la generazione dei segnali audio

Si deve ancora realizzare un blocco che avendo in ingresso i dati del generatore di sinusoidi (oltre ovviamente al clock a 50 Mhz ed al reset) generi in uscita tutti i segnali per pilotare il decoder.

In particolare questo dovrà generare:

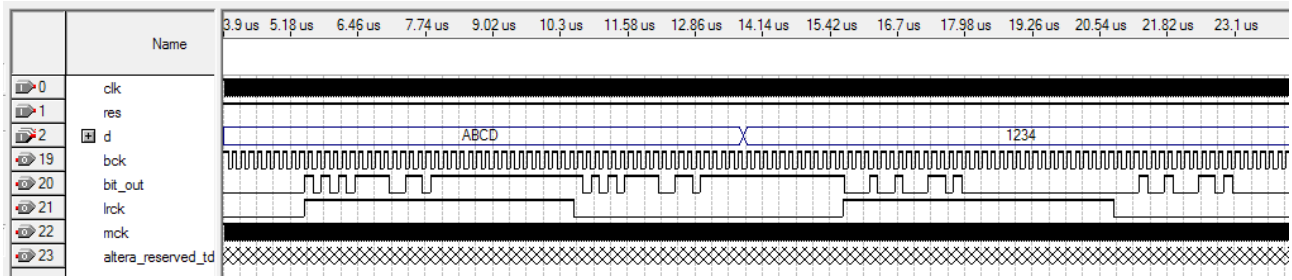
- Il master clock del decoder a circa 12.288 Mhz (nel nostro caso saranno 12.500 MHz)
  - 50 MHz / 2^2
- Il clock di alternanza L/R (a 48.8 KHz)
  - 50 MHz / 2^10
- Il Bit clock – supponendo di predisporre il sistema per gestire 32 bits per canale sarà a circa (3.12 MHz)
  - 50 MHz / 2^4
- Il flusso di dati audio seriali

Una possibile soluzione è qui di seguito rappresentata:

```
module gen_aud(d,clk_in,res,clk_mck,clk_bck,clk_lrck,bit_out);

input [15:0] d;
input clk_in;
input res;
output reg clk_mck;
output reg clk_bck;
output reg clk_lrck;
output reg bit_out;

reg [9:0] counter;
reg [4:0] bit_counter; // 32 bits

wire cont_mck=counter[0];
wire [2:0] cont_bck=counter[2:0];
wire [3:0] cont_bit=counter[3:0];
wire [8:0] cont_lrck=counter[8:0];
```

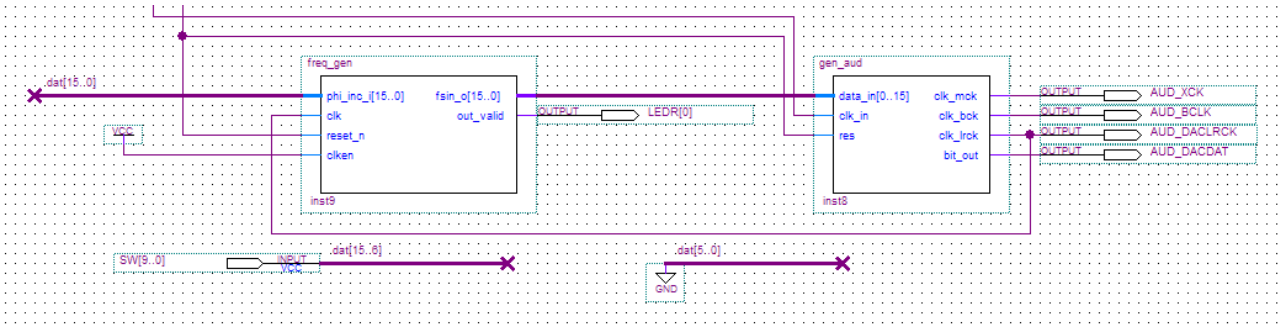
```
reg [15:0] data;


////////////   AUD Generator  //////////////
always@(posedge clk_in or negedge res)
begin
       if(!res)
        begin
              counter <=      8'b00000000;
              clk_mck <= 0 ; clk_bck <= 0 ;clk_lrck <= 0 ;
              bit_counter <= 4'b0000;
        end
       else
        begin
           counter <= counter + 8'b00000001;
              // MCLK  (12.5 MHz)
              if(cont_mck==1) clk_mck <=    ~clk_mck;
              // BCLK  (3.12  MHz) predisposto per 32 bit-canale
              if(cont_bck==7) clk_bck <=    ~clk_bck;
              // LRCLK  (48.8 KHz)
              if(cont_lrck==511)
                     begin

data[15:0]<={d[0],d[1],d[2],d[3],d[4],d[5],d[6],d[7],d[8],d[9],d[10],d[11],d[12],d[13],d[14],d[15]};
                       clk_lrck <= ~clk_lrck;
                       bit_counter <= 0;
                     end
              // bit counter
              if(cont_bit==0 & bit_counter <= 15)
                     begin
                       bit_counter <= bit_counter+1;
                       bit_out <= data[bit_counter];
                     end
        end
end

endmodule
```

Che può essere opportunamente simulato per verificarne il corretto funzionamento



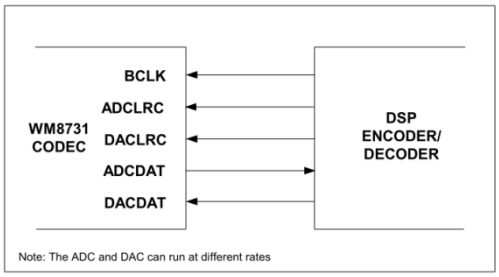
Si noti che la stringa seriale dei 16 bit componenti il messaggio da inviare risultano giustificati a sinistra e che tutti i bit oltre il 16imo (inutilizzati) replicano l'ultimo.

Si generi il simbolo corrispondente e successivamente si realizzi nello schematico (a livello Top Level) i seguenti collegamenti:



Si noti che mentre gli swithes pilotano i dieci bit più significativi del segnale di "fase_incrementale" (ovvero di pulsazione) i 6 bit meno significativi sono forzati al valore 0.
L'uscita `data_valid` è portata sul `LEDR[0]`

Si riconfiguri il decoder in modo tale che funzioni da Slave ed utilizzi il formato "left justified" a 16 bits (h0E01)



Si compili l'intero sistema e si effettui il download su DE1.

Si noti in particolare che poiché il progetto include un elemento (il generatore sinusoidale) il cui uso è concesso solo dietro licenza, peraltro assente nella versione "web", il sistema funziona a tempo indeterminato solo fintanto che il collegamento USB è attivato ed il relativo controllo è attivo. Nell'eventualità il collegamento USB venga staccato, il sistema funziona solo per un tempo limitato.

Nota:

Nel'uso del Signal Tap II – Logic Analizer, potrebbe essere comodo, ove si possieda la licenza, attivare il modo di funzionamento a "compilazione incrementale" che consentirebbe di risparmiare tempo in fase di compilazione. Capita talvolta che questo modo si attivi automaticamente, se in particolare non si possiede la licenza per questa opzione, la compilazione risulta impossibile. Per disattivare questa funzione si deve: digitare, nella finestra TCL il seguente comando:

Aprire la console tcl e digitare
View > Utility Wimdows > TCL console (Alt-2)
"set_global_assignment -name INCREMENTAL_COMPILATION OFF"